lesson3_full

Contents

LESSON 3 (FULL VERSION)

```
% reduce needless whitespace
format compact
% reduce irritations
more off
% start a diary
%diary lectureN.txt
```

THE PROBLEM WE WANT TO SOLVE

Assume that we have placed a hot bar with its ends in contact with ice water. The temperature of the bar will then decay over time to 0 degrees Centigrade. We have measured the temperature of the center of the bar at 6 times spaced half a minute apart. Taking the first of these times as time zero, the measured data are:

```
 time:           0      0.5    1     1.5    2     minutes
 Temperature:  14.60    8.42  4.86  2.80  1.62   Centigrade
```

We will define tMeasured and TMeasured as the six measured times and temperatures respectively.
Unknown to us, the exact temperature is given by

```
 TExact = 14.6 exp(- 1.1 t)
```

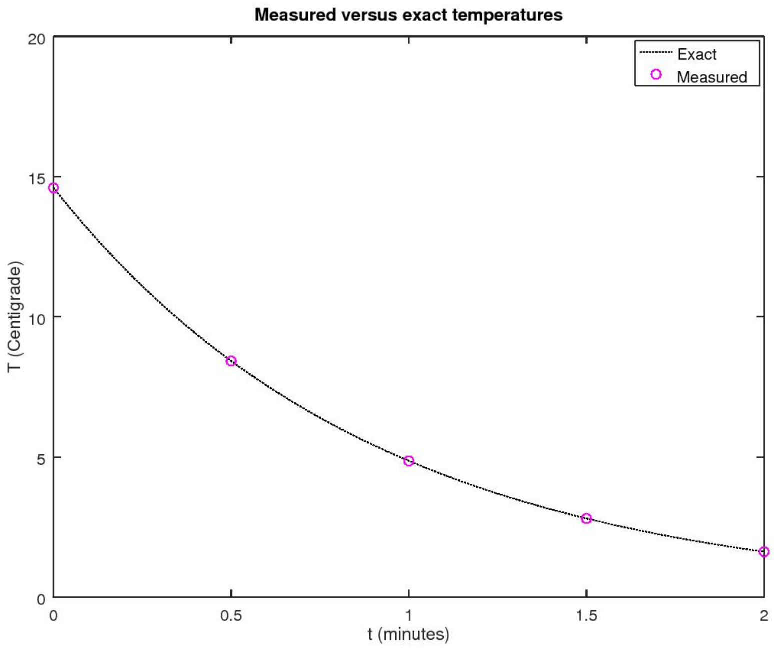
However, we only know the measured temperatures.
To make things easier, we will create a function TExactFun to evaluate the exact temperature that we pretend not to know. Since the function is very simple and not intended for more general use, we do not need to create an m file for it. Instead we can define TExactFun as a "handle" to an anonymous function.

```
% define tMeasured and TMeasured as given
tMeasured=[   0    0.5    1    1.5    2]';
TMeasured=[14.60  8.42  4.86  2.80  1.62]';

% define TExactFun as a handle to an anonymous function
TExactFun = @(t) 14.6*exp(-1.1*t);
```

PLOT TO UNDERSTAND THE PROBLEM BETTER

Let's plot the measured five values versus the exact solution that we pretend not to know.

```
% generate 100 time values between 0 and 2
tPlot=linspace(0,2,100)';
% generate corresponding exact temperatures
TExactPlot=TExactFun(tPlot);

% create the plot, using circles for the measured points
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om')
legend('Exact','Measured')
title('Measured versus exact temperatures')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```



INTERPOLATION

We would like to be able to evaluate the temperature at times in between the measured five times. This is called "interpolation".

For example, let's assume that we want to know the temperature at time 0.7, which is in between measured times 0.5 and 1.
Matlab provides 'interp1' or 'spline' to find it.

```
% let's evaluate T at t=0.7 using two different methods
t=0.7
TLinear=interp1(tMeasured,TMeasured,t)
TSpline=spline(tMeasured,TMeasured,t)

% two reasonable values, but which one is best???
TExact=TExactFun(t)
disp('For a nice smooth curve, spline interpolation is')
disp('much more accurate than linear interpolation!')
```

```
 t =   0.70000
 TLinear =   6.9960
 TSpline =   6.7513
 TExact =   6.7600
 For a nice smooth curve, spline interpolation is
 much more accurate than linear interpolation!
```

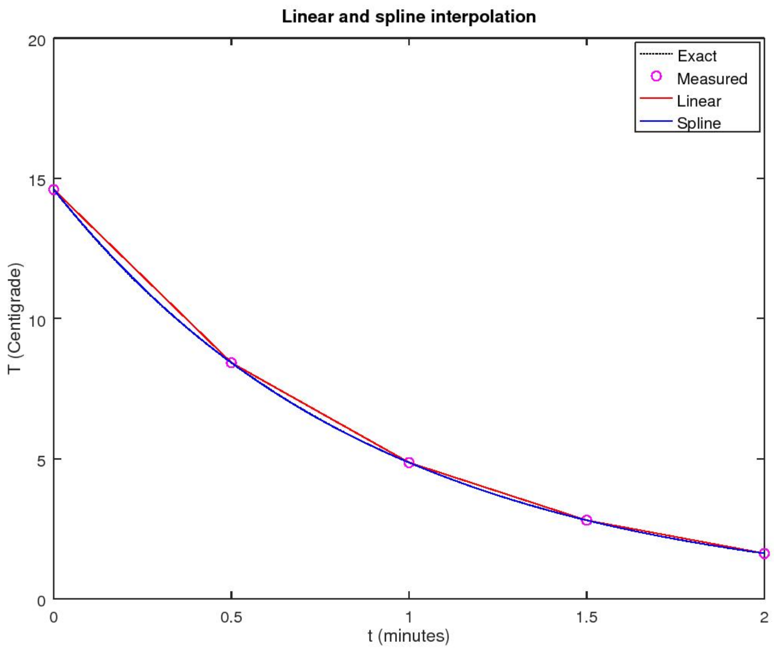
Compare the interpolations in a plot

```
% find the interpolated values at the plot times
TLinearPlot=interp1(tMeasured,TMeasured,tPlot);
TSplinePlot=spline(tMeasured,TMeasured,tPlot);

% compare the interpolations in a plot
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TLinearPlot,'r',...
     tPlot,TSplinePlot,'b')
legend('Exact','Measured','Linear','Spline')
title('Linear and spline interpolation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```

EXTRAPOLATION

Suppose that the time at which we want to know the temperature is t = 5. This time is not inside the measured range from 0 to 2. If that happens, we talk about *extrapolation* instead of *interpolation*.

```
 Extrapolation is much trickier than interpolation.
```



For that reason, interp1 refuses to do it unless you specify an additional "extrap" parameter.

```
% evaluate the values at t = 5
t=5
TLinear=interp1(tMeasured,TMeasured,t,'linear','extrap')
TSpline=spline(tMeasured,TMeasured,t)
TExact=TExactFun(t)
disp('Extrapolation is usually bad news!')

% Note that both linear and spline values are bad, and
% that the spline is much worse than linear.  But both
% values are useless.
```

```
t =  5
TLinear = -5.4600
TSpline = -14.700
TExact =  0.059667
Extrapolation is usually bad news!
```

NOISY DATA

What if the measured data have random errors? Suppose, for example, that the digital thermometer used to to measure the data only displays whole degrees C? Then the measured data:

```
Temperature: 14.60   8.42   4.86   2.80   1.62   Centigrade
```

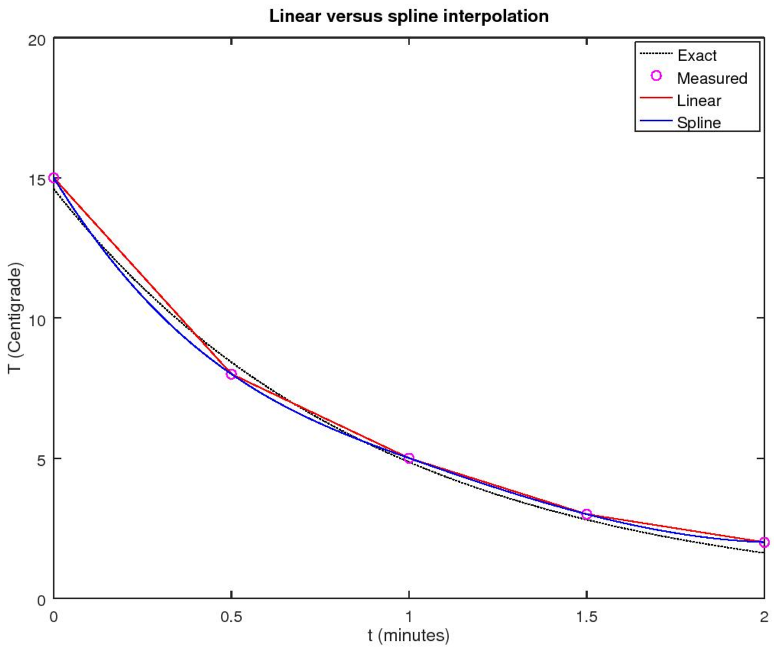
become:

```
Temperature: 15   8   5   3   2   Centigrade
```

Then what happens to our interpolations?

```
% correct the measured data list
TMeasured=[15 8 5 3 2]';

% interpolate again at 0.7
t=0.7
TLinear=interp1(tMeasured,TMeasured,t)
TSpline=spline(tMeasured,TMeasured,t)
TExact=TExactFun(t)
disp('Now the linear interpolation is actually better!');

% compare the interpolations in a plot
TLinearPlot=interp1(tMeasured,TMeasured,tPlot);
TSplinePlot=spline(tMeasured,TMeasured,tPlot);
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TLinearPlot,'r',...
     tPlot,TSplinePlot,'b')
legend('Exact','Measured','Linear','Spline')
title('Linear versus spline interpolation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% Because of the noise, the spline can be worse than
% linear.  The spline may also start oscilating if things
% get really bad.  Note the poor slope of the spline near
% time 2.

% compare the _maximum_ deviations
ErrLinear=max(abs(TLinearPlot-TExactPlot))
ErrSpline=max(abs(TSplinePlot-TExactPlot))
disp('There is no longer a real difference.')

% The maximum deviations are practically speaking the
% same.  There is no good reason to use spline
% interpolation instead of the simpler linear
% interpolation.
```

```
 t =  0.70000
 TLinear =  6.8000
 TSpline =  6.5300
 TExact =  6.7600
Now the linear interpolation is actually better!
 ErrLinear =  0.52550
 ErrSpline =  0.44453
 There is no longer a real difference.
```



CURVE FITTING

If we want to get a better predictions for the temperature given the noisy data, we must drop the assumption in interp1 and spline that the approximating curve goes to through all the measured points.
What we can do instead is find a relatively *simple* curve that is as close as possible to the data points. That is called "curve fitting".
In particular, recall that the exact temperature curve is given by

```
 TExact = 14.6 exp(- 1.1 t)
```

However, we are assuming that we do not know that. And given only our noisy data, there is *no way* to figure out that the above is the exact temperature.
But suppose that we assume (based on theoretical arguments not of importance here) that the desired temperature is of the form

```
T = A exp(B t)
```

and we then choose values for the constants A and B as well as we can based on our noisy data? That is likely to give a much better approximation than linear or spline interpolation.
Of course, the devil is in the details. Note first that with only 2 constants A and B and 5 noisy temperatures, there is no way that the expression above can reproduce all 5 noisy temperatures. You cannot solve 5 equations for 2 unknowns. You could select only 2 of the 5 temperatures and ignore the other 3, but which 2? If you are very lucky you could get a quite good approximation, but if you are not, you would get unnecessarily big errors.
It is a much better idea to use *all* the 5 data you have, and make TExpFit approximate them *on average* as well as it can. Typically, numerical analysist take "on average" to mean that they make the average *square* error as small as possible. There are both theoretical and practical reasons to do that. Theoretically, in simple cases where the errors are really random, this gives the best approximation possible. Practically, the mathematics of making the average *square* error as small as possible is a lot simpler than other possibilities (like making the maximum error as small as possible).
We do not really need to worry about the latter anyway, as Matlab does that work for us. What we should get away with is that what we are going to do is popularly known as the "Method of Least Squares". (Though "Method of Least Average Square Error" would be more accurate.)

Line fitting

Finding the best exponential approximation of the form

```
T = A exp(B t), call it TExpFit
```

is actually somewhat messy.
So, let's start more simple and try to find the best approximation of the data by a straight line,

```
T = C1 t + C2, call it TLinFit
```

Note that the expression above is *linear* in the coefficients C1 and C2 to find. That is unlike TExpFit where the coefficient B is inside an exponential, and then multiplied by A to boot. If the expression is linear in terms of the unknown coefficients, numerical analysists speak of "linear regression". That is another term you should try to remember.
Noting that a straight line is a polynomial of degree 1 (since the highest power of x is 1), we can first use Matlab function 'polyfit' to find the coefficients C1 and C2 and then 'polyval' to find the approximate temperature values

```
% find the coefficients C1 and C2 of the line
n=1;
CoefLin=polyfit(tMeasured,TMeasured,n)

% interpolate again at 0.7
t=0.7
TLinFit=polyval(CoefLin,t)
TExact=TExactFun(t)
disp('That is horrible!')

% let's see the linear fit in a plot
TLinFitPlot=polyval(CoefLin,tPlot);
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TLinFitPlot,'r')
legend('Exact','Measured','Linear fit')
title('Linear least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% print the error
ErrLinFit=max(abs(TLinFitPlot-TExactPlot))
disp('That is horrible, but what do you expect?')

% Clearly, a straight line cannot approximate the exact
% curve to a reasonable amount.
```

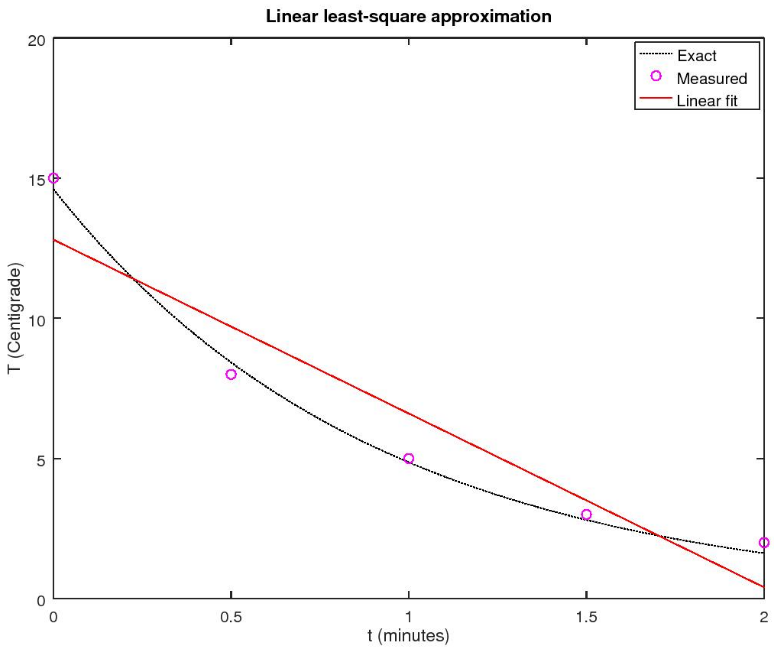
```
CoefLin =
   -6.2000   12.8000
t =  0.70000
TLinFit =  8.4600
TExact =  6.7600
That is horrible!
ErrLinFit =  1.8000
That is horrible, but what do you expect?
```

Fitting with a parabola

We can improve things quite a lot by approximating with a quadratic polynomial, i.e. a parabola,

```
T = C1 t^2 + C2 t + C3, call it TParFit
```

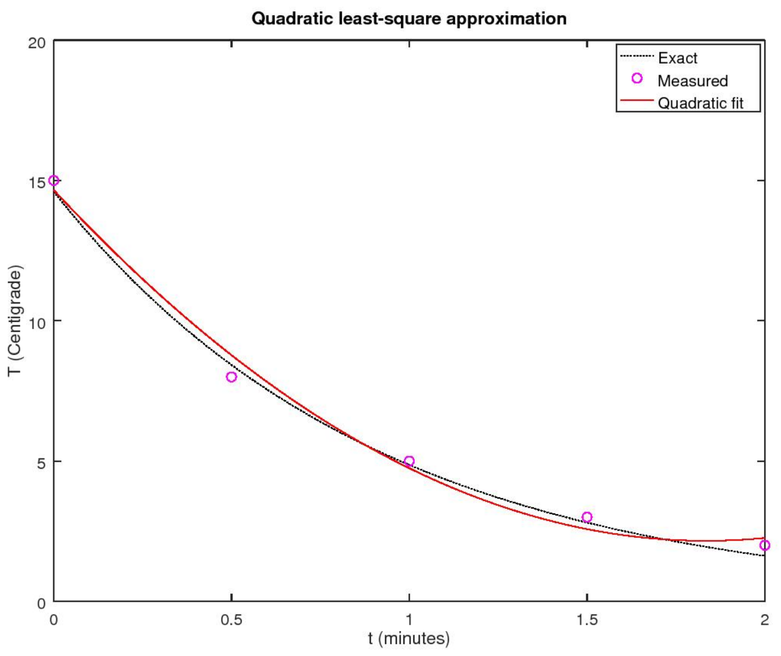
instead of a straight line.



```
% find coefficients C1, C2, and C3
n=2;
CoefPar=polyfit(tMeasured,TMeasured,n)

% interpolate again at 0.7
t=0.7
TParFit=polyval(CoefPar,t)
TExact=TExactFun(t)
disp('That is much better than the linear fit.')

% let's see the quadratic fit in a plot
TParFitPlot=polyval(CoefPar,tPlot);
plot(tPlot,TExactPlot,':k',...
      tMeasured,TMeasured,'om',...
      tPlot,TParFitPlot,'r')
legend('Exact','Measured','Quadratic fit')
title('Quadratic least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% print the error
ErrParFit=max(abs(TParFitPlot-TExactPlot))
```

```
% summarize the conclusion
disp('Not too bad, but the maximum error, at t=2, is
    quite big.')
```

```
CoefPar =
     3.7143   -13.6286    14.6571
t =  0.70000
TParFit =  6.9371
TExact =  6.7600
That is much better than the linear fit.
ErrParFit =  0.63942
Not too bad, but the maximum error, at t=2, is quite big
    .
```



Fitting with a quartic

Let's try fitting with a quartic,

```
TQuartFit = C1 x^4 + C2 x^3 + C3 x^2 + C4 x + C5
```

Note however, that now we are no longer *fitting*, but *interpolating*. With 5 unknown coefficients, the quartic can go through all 5 measured data points. This is usually a very bad idea.
In this particular case, the results below are much better than I expected. Fitting curves with too many coefficients can give very bad results. In this case the only real problem is the slope at t = 2. It might have been much worse.
The general rule of thumb is:

```
Do not interpolate a polynomial of degree more than
about the square root of the number of data points
```

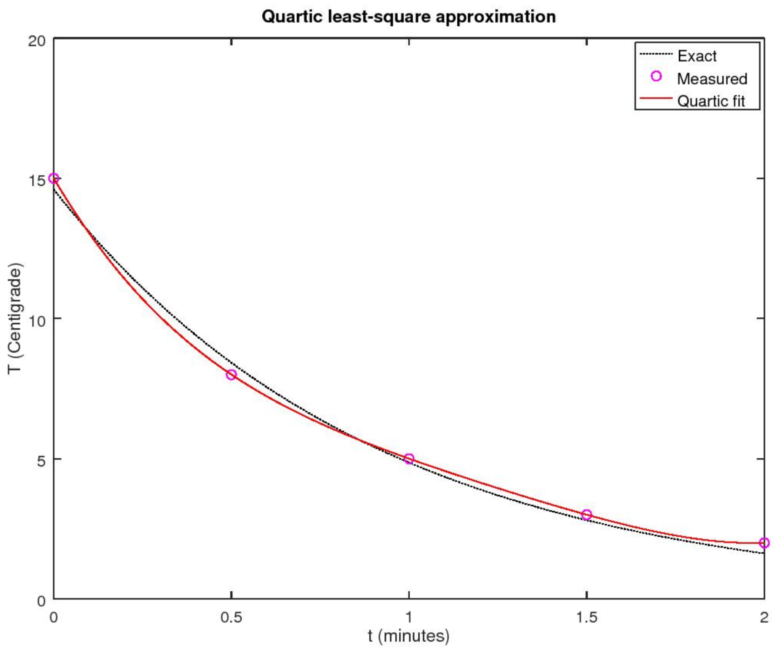
Since we have 5 data points and sqrt(5) is about 2, we should not fit a polynomial of a degree greater than 2. Exceptions confirm the rule.

```
% find the 5 coefficients
n=4;
CoefQuart=polyfit(tMeasured,TMeasured,n)

% interpolate again at 0.7
t=0.7
TQuartFit=polyval(CoefQuart,t)
TExact=TExactFun(t)

% let's see the quartic fit in a plot
TQuartFitPlot=polyval(CoefQuart,tPlot);
plot(tPlot,TExactPlot,':k',...
      tMeasured,TMeasured,'om',...
      tPlot,TQuartFitPlot,'r')
legend('Exact','Measured','Quartic fit')
title('Quartic least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% print the error
ErrQuartFit=max(abs(TQuartFitPlot-TExactPlot))
```

```
CoefQuart =
     2.0000   -10.0000    19.5000   -21.5000    15.0000
t =  0.70000
TQuartFit =  6.5552
TExact =  6.7600
ErrQuartFit =  0.47325
```

Fitting an exponential

Now, as explained earlier, we want to try to fit an exponential of the form



```
T = A exp(B t)
```

to our five data points. Hopefully this will give a better approximation to the exact temperature.
We can do this by taking a natural logarithm of the equation above:

```
ln(T) = ln(A) + B t
```

Defining

```
lnT = ln(T) = C1 t + C2
C1 = B
C2 = ln(A)
```

we can create a linear fit for lnT. Having lnT, we can simply take exponential to find T.
Alternatively, we could directly evaluate

```
TExpFit = exp(C1 t + C2)
```

but for now we will use polyfit and polyval on ln(T).
Note that Matlab uses log for ln (and log10 for log).

```
% create the measured lnT values
lnTMeasured=log(TMeasured);
```

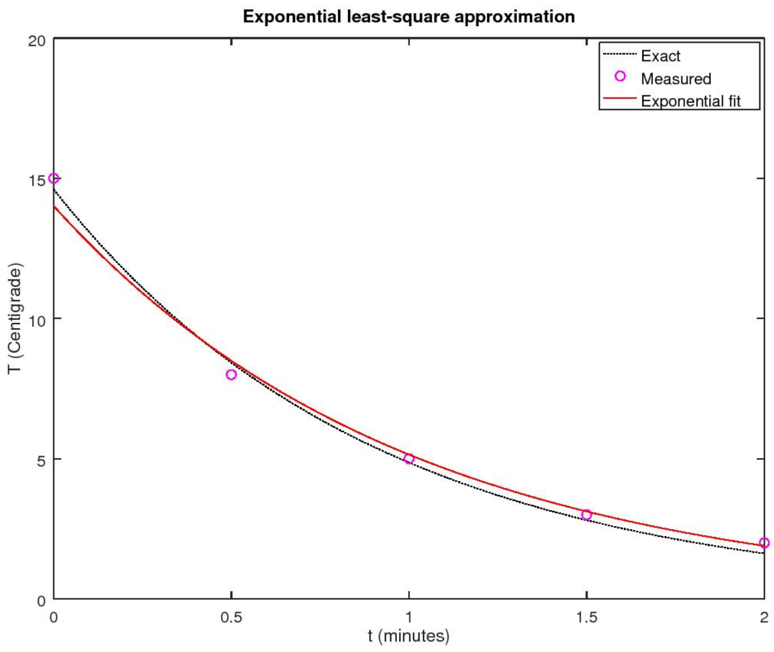
```
% find C1 and C2
n=1;
CoefExp=polyfit(tMeasured,lnTMeasured,n)

% interpolate again at 0.7
t=0.7
TExpFit=exp(polyval(CoefExp,t))
TExact=TExactFun(t)

% let's see the exponential fit in a plot
TExpFitPlot=exp(polyval(CoefExp,tPlot));
plot(tPlot,TExactPlot,':k',...
      tMeasured,TMeasured,'om',...
      tPlot,TExpFitPlot,'r')
legend('Exact','Measured','Exponential fit')
title('Exponential least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
ErrExpFit=max(abs(TExpFitPlot-TExactPlot))
disp('A bit disappointing, maybe.')
```

```
 CoefExp =
   -1.0021    2.6399
 t =  0.70000
 TExpFit =  6.9475
 TExact =  6.7600
 ErrExpFit =  0.58869
A bit disappointing, maybe.
```

Discussion

The reason that the maximum error in the exponential fit is not much better than the quadratic one has to do with our manipulations. Since we changed unknowns to ln(T), Matlab is no longer making the average error in T as small as possible. It is now making the average error in *ln(T)* as small as possible. This can be good or bad, depending on conditions. The error in T is the "absolute" error in the temperature. The error in ln(T) is the "relative" error in the temperature; the error relative to the magnitude of T. In other words, the error in ln(T) gives the percentage error in T. Sometimes you would rather have the relative error as small as possible, instead of the absolute error.

```
% let's check relative errors (note the ./)
RelErrParFit=max(abs(TParFitPlot-TExactPlot)./TExactPlot)
RelErrExpFit=max(abs(TExpFitPlot-TExactPlot)./TExactPlot)
```



```
% Note that the relative error in the exponential fit is
% much better than in the parabolic one.
```

```
RelErrParFit =   0.39526
RelErrExpFit =   0.16718
```

A dirty trick

If we are really interested in the absolute temperature in Kelvin, then the relative error in Centigrade is useless. In that case we really want the absolute error in temperature to be minimal. A trick works to get it while still using polyfit.
Note first that the intial temperature is about 6 times bigger than the final temperature. To force Matlab to give more attention to that larger value, we can put it inside the measured data lists 6 times. Similarly, the value at t = 0.5 is about 3 times the final value and the one at t = 1 about 2 times, So we place these data that many times in the measured data lists.

```
% tricked out data
tMeasuredMod = [ 0   0   0   0   0   0 0.5  0.5  0.5  1  1  1.5
    2]';
TMeasuredMod = [15  15  15  15  15  15   8    8    8   5  5   3
    2]';
```

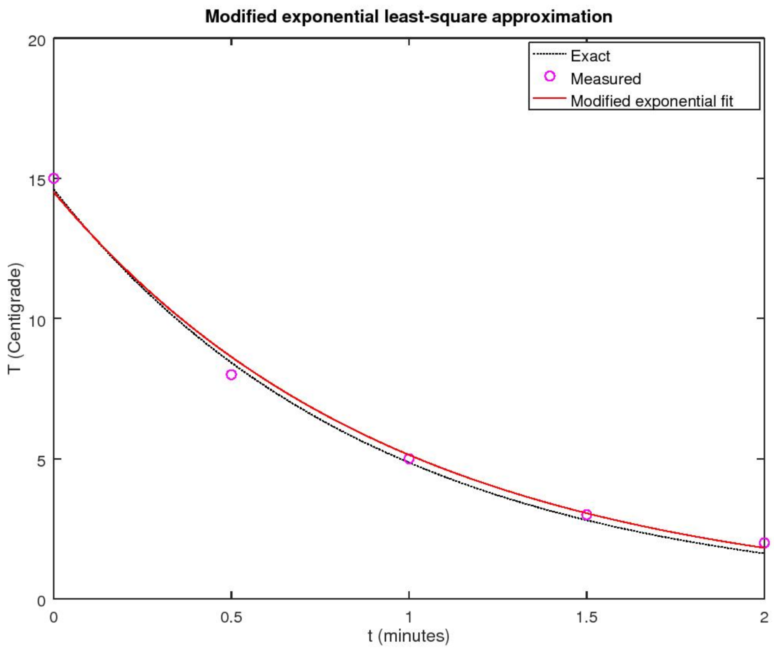
```
lnTMeasuredMod=log(TMeasuredMod);
n=1;
CoefExpMod=polyfit(tMeasuredMod,lnTMeasuredMod,n)

% interpolate again at 0.7
t=0.7
TExpFitMod=exp(polyval(CoefExpMod,t))
TExact=TExactFun(t)

% compare the interpolations in a plot
TExpFitModPlot=exp(polyval(CoefExpMod,tPlot));
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TExpFitModPlot,'r')
legend('Exact','Measured','Modified exponential fit')
title('Modified exponential least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
ErrExpFitMod=max(abs(TExpFitModPlot-TExactPlot))

% The error is now much smaller than anything seen
% before. And we also follow the slope of the exact
% curve very well.

% Another way to achieve a similar effect would be to
% concentrate the measurements near the start, where the
% temperature is largest.
```

```
CoefExpMod =
   -1.0388    2.6746
t =  0.70000
TExpFitMod =  7.0102
TExact =  6.7600
ErrExpFitMod =  0.27342
```

Extrapolation again

We already saw that extrapolation, i.e. evaluating outside the given range is fraught with peril. But the exponential fit is much less bad than the other methods.

```
% extrapolate again at t = 5
t=5
TLinear=interp1(tMeasured,TMeasured,t,'linear','extrap')
TSpline=spline(tMeasured,TMeasured,t)
TParFit=polyval(CoefPar,t)
```



```
TQuartFit=polyval(CoefQuart,t)
TExpFit=exp(polyval(CoefExp,t))
TExpFitMod=exp(polyval(CoefExpMod,t))
TExact=TExactFun(t)
```

```
t =   5
TLinear = -4
TSpline =  59
TParFit =  39.371
TQuartFit =  395.00
TExpFit =  0.093409
TExpFitMod =  0.080485
TExact =  0.059667
```

Derivatives

Sometimes we are interested in the derivative of the quantity in question. In the present example, it is a measure of how much heat leaks out of the bar per unit time.
Since

```
TExact = 14.6 exp (- 1.1 t)
```

its derivative is simply

```
- 1.1 TExact
```

(That follows from differentiating the exponential using the chain rule.)
Similar for

```
TExpFit = exp(CoefExp(1)*t + CoefExp(2))
```

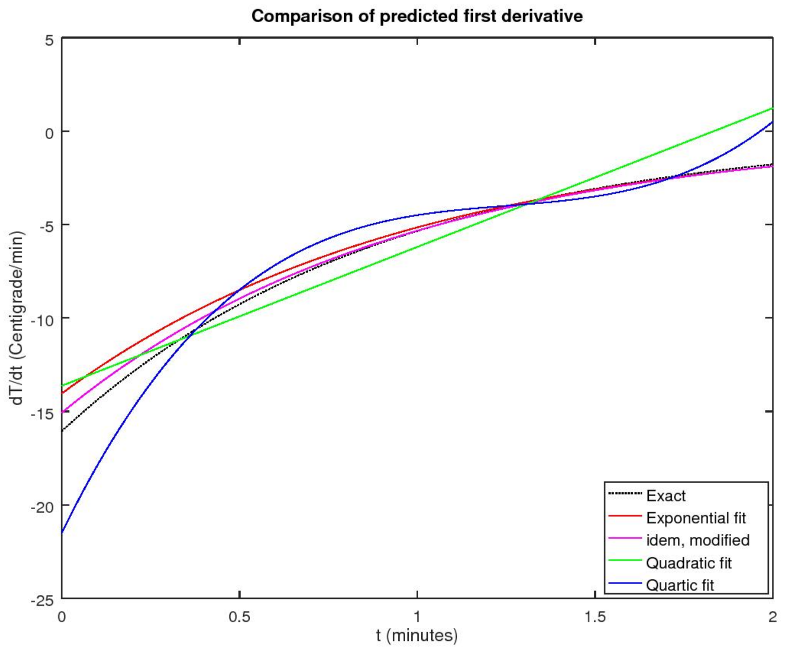
the derivative is

```
CoefExp(1) TExpFit.
```

For the linear, quadratic, and quartic fits, we can use the fact that 'polyder' will find the coefficients of the derivative polynomial for us.
How about the derivative of your beloved interpolated spline? Well, linear and spline interpolation are described by "piecewise polynomials": there is a different polynomial in each segment between measured points. The bad thing is that the idiots at MathWorks never defined a function to find the derivatives of piecewise polynomials. If you want the derivative of your spline, look for 'ppder' or 'ppdiff' provided by third parties, (where pp is an acronym for "piecewise polynomial".) (Octave provides ppder.)

```
% derivative of TExact
derTExactPlot=-1.1*TExactPlot;

% derivative of TExpFit
derTExpFitPlot=CoefExp(1)*TExpFitPlot;
derTExpFitModPlot=CoefExpMod(1)*TExpFitModPlot;

% derivative of TParFit
derCoefPar=polyder(CoefPar);
derTParFitPlot=polyval(derCoefPar,tPlot);

% derivative of TQuartFit
derCoefQuart=polyder(CoefQuart);
derTQuartFitPlot=polyval(derCoefQuart,tPlot);

% plot it
plot(tPlot,derTExactPlot,':k',...
     tPlot,derTExpFitPlot,'r',...
     tPlot,derTExpFitModPlot,'m',...
     tPlot,derTParFitPlot,'g',...
     tPlot,derTQuartFitPlot,'b')
legend('Exact','Exponential fit','idem, modified',...
       'Quadratic fit','Quartic fit')
title('Comparison of predicted first derivative')
xlabel('t (minutes)')
```

```
ylabel('dT/dt (Centigrade/min)')
legend('location','southeast')
```



Integrals

Suppose we want to know how much radiation the bar emits per unit surface area while cooling down. Assuming that the bar surface is perfectly black, the Stefan-Boltzmann law says that the radiation emitted per unit area and unit time is given by

```
qdot = sigma T^4
```

where sigma is the Stefan-Boltzmann constant with value given below, and T is the absolute temperature.
So to get the desired radiation q, given our temperature,

- convert Centigrade to Kelvin by adding T0 = 273.15
- raise that to power 4 (square twice)
- integrate that from t = 0 to t = 2
- multiply by 60 to convert dt to seconds

- multiply by sigma (let's do that last)

Matlab can do the integration of the temperatures for us using 'integral' if we provide the function to integrate. Or it can do it using 'polyint' if we provide it the coefficients of the polynomial to integrate.
Note: Octave still uses the old name 'quad' instead of 'integral'.
Note: Since TExactFun is *not* a polynomial, and neither is TExpFit, we cannot use polyint on these two.
Note: Our polynomial expressions for the square and quartic fits give the temperature T as a polynomial:

```
T = C1 T^2 + C2 T + C3
T = C1 T^4 + C2 T^3 + C3 T^2 + C4 T + C5
```

To convert to Kelvin, we can add 273.15 to the final, 'end', coefficient C3 repectively C5. But to get the temperature to the power 4, we must multiply it by itself, then multiply **that** by itself. To multiply two polynomials, use the 'conv' function. That will find the coefficients of the product polynomial. It is called 'conv' because the people at MathWorks do not have enough brains to define 'polymul' as an alias for 'conv'.
Note: An alternative approach would be to find the fourth power of the measured temperatures in Kelvin, then approximate that by a polynomial from scratch. I believe this would be less accurate, but you can try it out for yourself.
Note: to find the exact integral of your beloved spline, look for a third-party 'ppint' or similar. (Octave provides ppint.) Or use numerical integration as below.
Note that **all** values below are pretty accurate. This is typical:

```
    Numerical errors tend to become less important
  in integrals, and more important in derivatives.
```

```
% the Stefan-Boltzmann constant in W/m^2 K^4:
sigma=5.670373E-8;

% 0 degrees Centigrade in Kelvin
T0=273.15;

% Let's find the exact integral first using our
% knowledge of Calculus I.  To integrate
%        integral (A exp(Bt)+T0)^4 dt
% change variable to u = A exp(Bt), etc:

% names for the constants in TExact = A exp(B t)
A=14.6;
B=-1.1;
```

```
% evaluate the end values of u
u1=A;
u2=A*exp(B*2);

% evaluate the integral as found by calculus
qExactExact=(...
    1/4*(u2^4-u1^4)+...
    4/3*T0*(u2^3-u1^3)+...
    3*T0^2*(u2^2-u1^2)+...
    4*T0^3*(u2-u1)+...
    T0^4*(log(u2)-log(u1)))/B*60*sigma

% Next let's use numerical integration, i.e. 'integral'
% or 'quad', to find the integral. Note that there will
% be an error created by the numerical integration, even
% if we integrate the exact temperature.

% try numerical integration of the exact temperature
qExactNum=quad(@(t) (TExactFun(t)+T0).^4,0,2)*60*sigma
disp('As shown, numerical integration for a smooth')
disp('function like this will be very accurate.')
disp('The error is smaller than the round-off.')

% try numerical integration of the linear interpolation
qLinNum=quad(@(t) (interp1(tMeasured,TMeasured,t)+T0)
    .^4,0,2)*60*sigma

% try numerical integration of the spline interpolation
qSplineNum=quad(@(t) (spline(tMeasured,TMeasured,t)+T0)
    .^4,0,2)*60*sigma

% try numerical integration of the parabolic fit
qParNum=quad(@(t) (polyval(CoefPar,t)+T0).^4,0,2)*60*
    sigma

% try numerical integration of the quartic fit
qQuartNum=quad(@(t) (polyval(CoefQuart,t)+T0).^4,0,2)*60*
    sigma

% try numerical integration of the exponential fit
B=CoefExp(1);
lnA=CoefExp(2);
qExpFit=quad(@(t) (exp(B*t+lnA)+T0).^4,0,2)*60*sigma

% try numerical integration of the tricked one too
B=CoefExpMod(1);
```

```
lnA=CoefExpMod(2);
qExpFitMod=quad(@(t) (exp(B*t+lnA)+T0).^4,0,2)*60*sigma

% For the polynomial fits, however, we can integrate *
    exactly* using
% polyint. Not only does this eliminate all integration
    errors,
% however small, Matlab can also do it much more quickly.
      That could
% be important if you have to do a lot of these integrals
    .

% first convert the polynomial to Kelvin by adding T0
tempC=CoefPar;
tempC(end)=tempC(end)+T0;

% now square that polynomial twice to get T^4
tempC=conv(tempC,tempC);
tempC=conv(tempC,tempC);

% find the antiderivative polynomial using polyint
tempC=polyint(tempC);

% evaluate the integral between 0 and 2 with that
tempInt=polyval(tempC,2)-polyval(tempC,0);

% add the remaining factors
qParExact=tempInt*60*sigma

% We can integrate the quartic fit the same way

% first convert the polynomial to Kelvin by adding T0
tempC=CoefQuart;
tempC(end)=tempC(end)+T0;

% now square that polynomial twice to get T^4
tempC=conv(tempC,tempC);
tempC=conv(tempC,tempC);

% find the antiderivative polynomial using polyint
tempC=polyint(tempC);

% evaluate the integral between 0 and 2 with that
tempInt=polyval(tempC,2)-polyval(tempC,0);

% add the remaining factors
```

```
qQuartExact=tempInt*60*sigma
```

```
qExactExact =      4.1302e+04
qExactNum =      4.1302e+04
As shown, numerical integration for a smooth
function like this will be very accurate.
The error is smaller than the round-off.
qLinNum =      4.1434e+04
qSplineNum =      4.1307e+04
qParNum =      4.1352e+04
qQuartNum =      4.1297e+04
qExpFit =      4.1384e+04
qExpFitMod =      4.1422e+04
qParExact =      4.1352e+04
qQuartExact =      4.1297e+04
```

MORE MEASUREMENTS

If we would measure a lot more points than the five we have, and the errors in these measurements would be random, we could get a much better approximation.
Unfortunately, rounding of temperatures to whole degrees is **not** random. It creates a deterministic "staircase" of numbers. But we can try anyway.

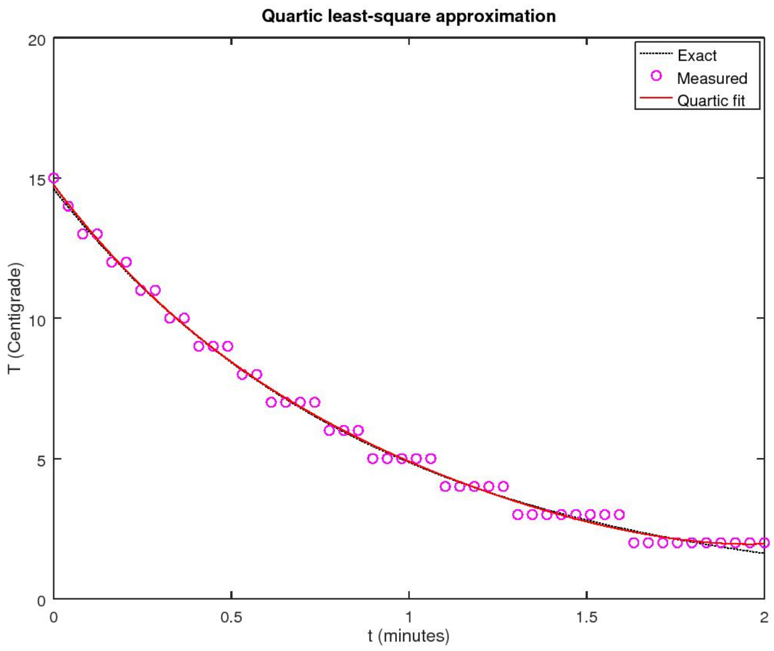
```
% Note: We will cheat, and use the exact solution, which
% we are not supposed to know, to avoid doing and typing
% in 50 measurements.

% create the new "measured" data
tMeasured=linspace(0,2,50)';
% 'round' rounds to whole numbers
TMeasured=round(TExactFun(tMeasured));

% use some more plot points now
tPlot=linspace(0,2,200)';
TExactPlot=TExactFun(tPlot);
```

Quartic fit with more noisy data

```
% repeat the quartic fit
n=4;
CoefQuart=polyfit(tMeasured,TMeasured,n)
```

```
% compare the interpolations in a plot
TQuartFitPlot=polyval(CoefQuart,tPlot);
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TQuartFitPlot,'r')
legend('Exact','Measured','Quartic fit')
title('Quartic least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
ErrQuartFit=max(abs(TQuartFitPlot-TExactPlot))
```

```
CoefQuart =
     0.87772    -4.44749    10.65526   -16.93789    14.74753
ErrQuartFit =  0.33872
```



Quintic fit with more noisy data

With the additional data, we can try a quitic (5th degree) fit now.
But do not even **think** about interpolating a polynomial of degree 49. It would just crash.

```
% create the fit
```

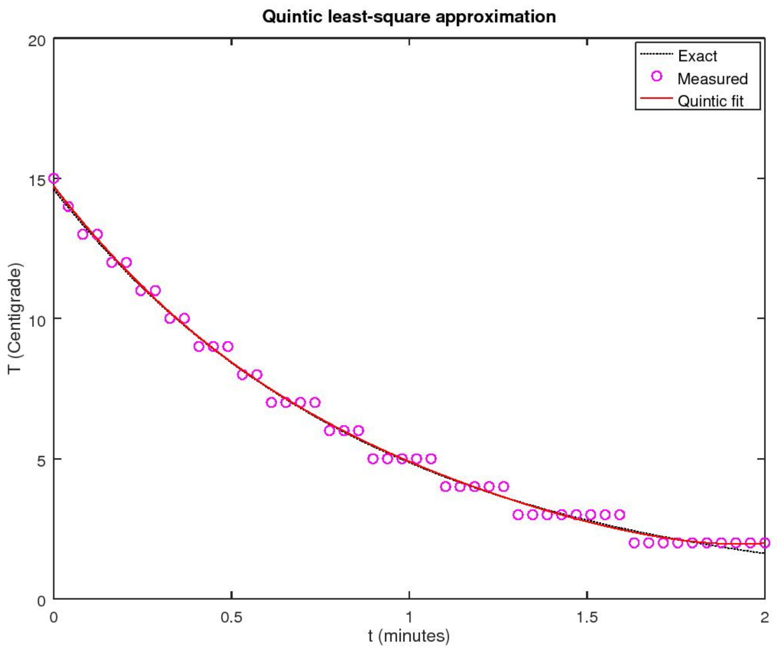
```
n=5;
CoefQuint=polyfit(tMeasured,TMeasured,n)

% compare the interpolations in a plot
TQuintFitPlot=polyval(CoefQuint,tPlot);
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TQuintFitPlot,'r')
legend('Exact','Measured','Quintic fit')
title('Quintic least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
ErrQuintFit=max(abs(TQuintFitPlot-TExactPlot))
disp('The disappointing result is due to the fact that')
disp('The final "measured" points are all too high.')
disp('Have a good look at the end of the graph!')
```

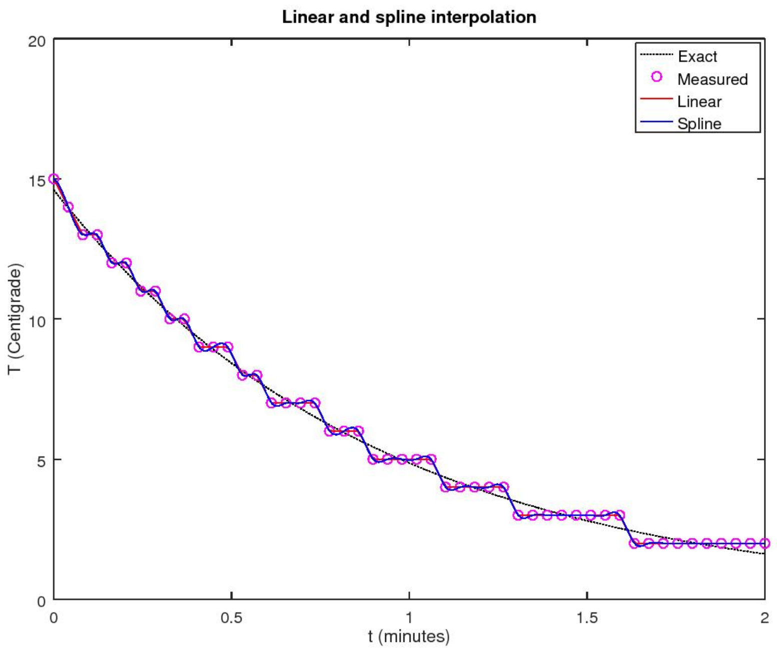
```
CoefQuint =
     0.27835    -0.51404    -1.98510     8.83513   -16.43811
          14.71890
ErrQuintFit =  0.36735
The disappointing result is due to the fact that
The final "measured" points are all too high.
Have a good look at the end of the graph!
```

Interpolation with more noisy data

Note also that interpolations would *not* improve if we used more points.

```
% compare the interpolations in a plot
TLinearPlot=interp1(tMeasured,TMeasured,tPlot);
TSplinePlot=spline(tMeasured,TMeasured,tPlot);
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TLinearPlot,'r',...
     tPlot,TSplinePlot,'b')
legend('Exact','Measured','Linear','Spline')
title('Linear and spline interpolation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% Note that here the spline interpolation is definitely
% worse than linear, though not by much.  (Except if you
% looked at the derivative.)
```


Quintic least-square approximation
Exact
Measured
Quintic fit
T (Centigrade)
t (minutes)


Linear and spline interpolation
Exact
Measured
Linear
Spline
T (Centigrade)
t (minutes)

The exponential fit with more noisy data

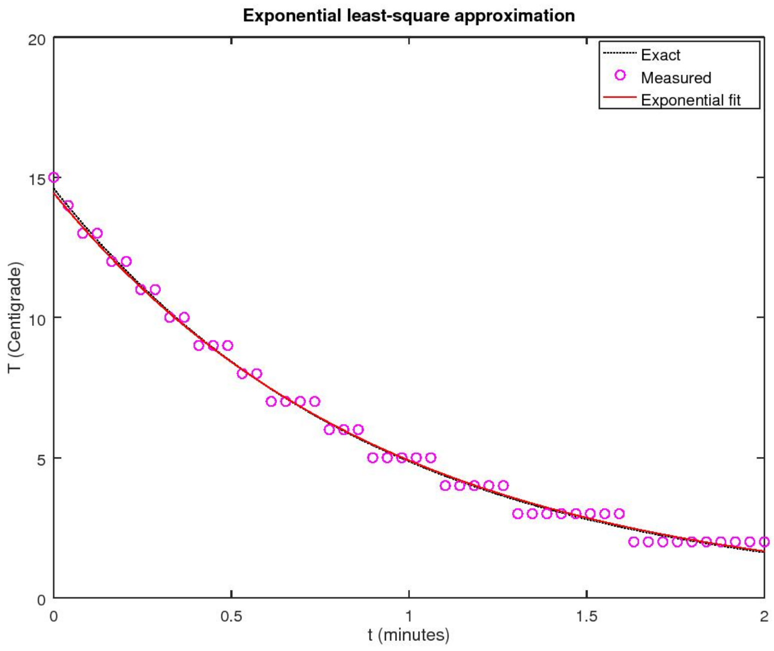
```
% repeat the exponential fit
lnTMeasured=log(TMeasured);
n=1;
CoefExp=polyfit(tMeasured,lnTMeasured,n)

% compare the interpolations in a plot
TExpFitPlot=exp(polyval(CoefExp,tPlot));
plot(tPlot,TExactPlot,':k',...
     tMeasured,TMeasured,'om',...
     tPlot,TExpFitPlot,'r')
legend('Exact','Measured','Exponential fit')
title('Exponential least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
ErrExpFit=max(abs(TExpFitPlot-TExactPlot))
disp('Clearly, that is very good.')
```

```
CoefExp =
  -1.0808    2.6699
ErrExpFit =  0.16097
Clearly, that is very good.
```

ADDITIONAL REMARKS

Often you would want your spline to satisfy end conditions. For example, you might want it to have given derivatives at the ends. Or be periodic. Given derivatives at the ends can be achieved using 'spline' if you add the desired two values to the function values list. For more complicated cases, consider function 'csape'.

End lesson 3


Exponential least-square approximation
Exact
Measured
Exponential fit
T (Centigrade)
t (minutes)
0
5
10
15
20
0
0.5
1
1.5
2